PCI-to-PCI Bridge Architecture Specification

Revision 1.1

December 18, 1998

Revision History

Revision	Issue Date	Comments
1.0	4/5/94	Original issue
1.1	12/18/98	Update to include target initial latency requirements.

The PCI Special Interest Group disclaims all warranties and liability for the use of this document and the information contained herein and assumes no responsibility for any errors that may appear in this document, nor does the PCI Special Interest Group make a commitment to update the information contained herein.

Contact the PCI Special Interest Group office to obtain the latest revision of the specification.

Questions regarding the PCI specification or membership in the PCI Special Interest Group may be forwarded to:

PCI Special Interest Group
2575 N.E. Kathryn #17
Hillsboro, Oregon 97124
1-800-433-5177 (USA)
503-693-6232 (International)
503-693-8344 (Fax)
pcisig@pcisig.com
http://www.pcisig.com

DISCLAIMER

This PCI-to-PCI Bridge Architecture Specification is provided "as is" with no warranties whatsoever, including any warranty of merchantability, noninfringement, fitness for any particular purpose, or any warranty otherwise arising out of any proposal, specification, or sample. The PCI SIG disclaims all liability for infringement of proprietary rights, relating to use of information in this specification. No license, express or implied, by estoppel or otherwise, to any intellectual property rights is granted herein.

Intel and Pentium are registered trademarks of Intel Corporation.

All other product names are trademarks, registered trademarks, or servicemarks of their respective owners.

CONTENTS

CHAPTER 1 INTRODUCTION

CHAPTER 2 BRIDGE REQUIREMENTS

CHAPTER 3 CONFIGURATION

CHAPTER 4 ADDRESS DECODING

CHAPTER 5 BUFFER MANAGEMENT

CHAPTER 12 VGA SUPPORT

CHAPTER 13 SLOT NUMBERING

Preface

Scope

This specification defines the behavior of a compliant PCI-to-PCI bridge. A PCI-to-PCI bridge that conforms to this specification and the *PCI Local Bus Specification* is a compliant implementation. Compliant bridges may differ from each other in performance and to some extent functionality.

Related Documents

This specification assumes that the reader has a working knowledge of the *PCI Local Bus Specification* and is familiar with other PCI specifications. Refer to the PCI SIG web page for the latest list of specifications and revision levels.

Following publication of the *PCI-to-PCI Bridge Architecture Specification*, there may be future approved errata and/or approved changes to the specification prior to the issuance of another formal revision. To assure designs meet the latest level requirements, designers of PCI-to-PCI bridges must refer to the PCI SIG home page at http://www.pcisig.com, in the members-only section, for any approved changes.

Chapter 5
Buffer Management

5.1. Prefetching Read Data

The term prefetch is used when the bridge reads data from the target in anticipation that the master will consume it. Prefetching is a useful technique for hiding the latency of a burst read transaction but its use is restricted. Memory that is prefetchable has the attribute that it returns the same data when read multiple times and does not alter any other device state when it is read[3]. When prefetching, the bridge may read data that is not consumed by the master. The bridge is required to discard any prefetched read data not consumed when the master concludes the read transaction (refer to Section 5.6.2.).

The *PCI Local Bus Specification* specifies that a bridge may safely prefetch data when the transaction uses the Memory Read Line or Memory Read Multiple command. Since most processor architectures do not have the notion of prefetchable memory, typical host bus bridges do not generate Memory Read Line or Memory Read Multiple transactions. A bridge may provide an optional address range that allows the bridge to prefetch memory read data from a target attached to the secondary interface of the bridge. A target explicitly indicates to configuration software that a memory address range is prefetchable by setting the Prefetchable bit (bit 3) in the corresponding Base Address Register (BAR) within the target's Configuration Space header. Configuration software uses this information to program the prefetchable memory address range in the bridge and to map the prefetchable address ranges of secondary bus targets into the range. If a master on the primary interface of a bridge accesses a location mapped in the prefetchable memory address range, the bridge is permitted to prefetch read data when completing the transaction on the secondary bus. In this case, the bridge is permitted either to extend the read transaction burst length or to modify the bus command to Memory Read Line or Memory Read Multiple or both. This function is supported only if the optional Prefetchable Memory Base and Limit registers are implemented.

A bridge is also permitted to prefetch data from a secondary bus if the transaction originates on the primary bus and is either a Memory Read Line or Memory Read Multiple command. A

[3] In a prior revision of this specification, a PCI-PCI bridge was not permitted to prefetch read data across a 4 KB boundary. However, this restriction has been removed by this specification revision. It is the responsibility of the target device to disconnect a burst transaction when either a BAR boundary is reached, or a boundary is reached within a BAR where the attributes of the access change (i.e., prefetchable vs. non-prefetchable).

bridge is permitted to prefetch data from the primary bus if the transaction originates on the secondary bus and is either a Memory Read Line or Memory Read Multiple command. Masters attached to the secondary interface of a PCI-to-PCI bridge are encouraged to use the Memory Read Line or Memory Read Multiple commands if they desire high performance (prefetchable) read transactions. The bridge is also permitted to assume that all transactions that originate on the secondary bus and go up through the bridge have a final destination at main memory and therefore are prefetchable. If a bridge makes this assumption and does blind prefetching on the Memory Read command, it must support a device-specific bit (in Configuration Space) that allows this feature to be disabled (if blind prefetching causes a problem). When prefetching memory read data, the bridge is permitted to assert all byte enables for all data phases on the destination bus independent of the byte enables used by the originating bus master.

Table 5-1 lists when prefetching by the bridge is permitted for those bus commands that function like a read transaction (i.e., the master is reading data from the target device). Prefetching does not apply for those bus commands where the master writes data to the target device, the Dual Address Command encoding, or for reserved bus command encodings.

Table 5-1: Read Prefetch Summary

C/BE[3::0]#	Command type	Access Originates on the:	
		Primary Bus	Secondary Bus
0010	I/O Read	No	No
0110	Memory Read	No (Note 1)	No (Note 2)
1010	Configuration Read	No	No
1100	Memory Read Multiple	Yes	Yes
1110	Memory Read Line	Yes	Yes

Notes:

1. Yes when the address is in the prefetchable range as described by Prefetchable Memory Base and Limit registers.
2. The bridge is permitted to treat this like Memory Read Line or Memory Read Multiple, but this feature must be able to be turned off via a device-specific bit.

Prefetching of read data is never allowed in the PCI I/O Space or Configuration Space.

5.2. Posting Write Data

Posting of write data is required by the *PCI Local Bus Specification* if either Memory Write or Memory Write and Invalidate commands are used for transactions that cross the bridge in either direction and the bridge has posting buffer space available. Posting of I/O Write and Configuration Write transactions is not permitted by a bridge.

A PCI-to-PCI bridge is generally allowed to terminate with Retry a transaction that uses the Memory Write or Memory Write and Invalidate commands only when its buffers are filled with previously received memory write data or for a locked operation. Terminating a Memory Write or Memory Write and Invalidate transaction with Retry for other reasons can lead to deadlocks. Refer to Section 5.6.3. for more details.

5.2.1. Memory Write and Invalidate Usage

The *PCI Local Bus Specification* permits a master to use the Memory Write and Invalidate (MWI) command to transfer memory write data when the following requirements are met:

- linear incrementing address mode is used
- the transaction begins on a cacheline aligned boundary
- the master guarantees that it will deliver all bytes within any cacheline accessed (in some cases the transaction will be a burst transaction which accesses multiple cachelines)

When functioning as a bus master a bridge is permitted to use the MWI command when forwarding transactions (upstream or downstream) in one of three cases:

- the bridge forwards an MWI transaction originated by another master
- the bridge promotes a Memory Write (MW) transaction (or portion of a MW transaction) to a MWI transaction
- the bridge combines sequential MW transactions to generate a transaction that meets the MWI usage requirements

Note that for each case there are requirements that qualify the bridges ability to use the MWI command for the transaction. Each case is described in detail in the following sections.

5.2.1.1. Forwarding Memory Write and Invalidate Transactions

A bridge is permitted to forward a MWI transaction originated by another master to the opposite interface as a MWI transaction when the following conditions are true:

- the bridge implements the Cacheline Size register (see Section 3.2.4.7.)
- the Cacheline Size register has been set to a value supported by the bridge

The bridge is not required to validate that the forwarded transaction meets the MWI usage requirements (the originating master is responsible for meeting the MWI usage requirements). The setting of the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) does

not affect the bridge's ability to use the MWI command on the destination bus when forwarding a MWI transaction originated by another master.

If the bridge does not implement the Cacheline Size register, or the Cacheline Size is not set to a value supported by the bridge, then the bridge must change the MWI command to MW when forwarding the transaction across the bridge[4].

5.2.1.2. Promoting Memory Write Transactions

When forwarding a MW transaction, the bridge can optionally promote the transaction to a MWI transaction if it meets the MWI usage rules. In this case, the bridge is required to validate that the forwarded transaction meets all MWI usage requirements. To use this method, the bridge must implement the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) and the Cacheline Size register (see Section 3.2.4.7.). Additionally, the Memory Write and Invalidate bit must be set to enable the bridge promote Memory Write transactions to MWI transactions.

A bridge is permitted to meet the MWI usage rules by promoting only a subset of a MW transaction. For example, consider a MW burst transaction that begins and ends on address boundaries that are not cacheline aligned but that contain one or more cachelines within the burst that have all bytes enabled. When forwarding the MW transaction, the bridge is permitted to meet the MWI usage requirements by segmenting the original MW transaction into three separate transactions on the destination bus. The first transaction would use a MW transaction to transfer the memory write data from the starting address (which is not cacheline aligned) up to the next aligned cacheline boundary. The bridge would then use a MWI transaction to transfer the memory write data beginning on the aligned cacheline boundary including all subsequent complete cachelines up to the final aligned cacheline boundary contained in the original MW transaction. The bridge would then use a MW transaction to transfer the remainder of the data contained in the original MW transaction. Note that the bridge cannot alter the ordering of the original MW transaction. All bytes must be forwarded in the order they were received in the original MW transaction.

5.2.1.3. Combining Memory Write Transactions

A bridge may optionally combine sequential Memory Write transactions (see Section 5.7.) to generate a transaction that meets MWI usage requirements. In this case, the bridge is required to validate that the forwarded transaction meets all MWI usage requirements. To use this method, the bridge must implement the Memory Write and Invalidate bit in the Command register (see Section 3.2.4.3.) and the Cacheline Size register (see Section 3.2.4.7.). Additionally, the Memory Write and Invalidate bit must be set to enable the bridge to combine Memory Write transactions into MWI transactions. Note that the bridge cannot alter the ordering of the original MW transaction. All bytes must be forwarded in the order they were received in the original MW transaction.

---

[4] A valid Cacheline Size is necessary for the bridge to determine cacheline boundaries on the target bus when the Master Latency Timer on the destination bus expires during the delivery of a MWI transaction (see Sections 3.2.4.8. and 3.2.5.5.).

5.2.1.4. Memory Write and Invalidate Disconnects

The *PCI Local Bus Specification* permits a target to disconnect a Memory Write and Invalidate transaction on an address that is not an aligned cacheline boundary. In this case, when the master resumes delivery of the remaining write data for the cacheline in which the disconnect occurred, the master must use a Memory Write transaction (the Memory Write and Invalidate usage requirements are no longer valid). The following sections describe the requirements for delivering the remainder of a MWI transaction for two cases of Target-Disconnect:

- the bridge disconnects the originating master on the originating bus
- the target disconnects the bridge on the destination

These two cases are discussed in the following sections.

5.2.1.4.1. Master Disconnected by the Bridge

When responding as a target, a bridge can disconnect a master during any data phase of a MWI transaction (this may occur as the result of a temporary buffer full condition for example). The bridge must use a MW transaction on the destination bus to deliver the write data for the incomplete cacheline in which the disconnect occurred. The bridge is permitted to use the MWI command to forward any cachelines posted by the bridge *prior* to signaling disconnect to the originating master provided the requirements detailed in Section 5.2.1.1. are met.

5.2.1.4.2. Bridge Disconnected by the Target

When a bridge is disconnect by the target when forwarding a MWI transaction on an address boundary that is not cacheline aligned, the bridge must use a MW transaction to deliver the remaining write data for the cacheline in which the disconnect occurred. The bridge is permitted to use the techniques described in Sections 5.2.1.2. and 5.2.1.3. to resume delivery of subsequent cachelines with MWI transactions.

5.3. Delayed Transactions

The following discussion of Delayed Transactions is included to clarify their application to PCI-to-PCI bridges. For a complete treatment of Delayed Transactions, refer to the *PCI Local Bus Specification*.

Bridges must implement Delayed Transactions to meet the latency requirements of the *PCI Local Bus Specification*. Delayed Transactions significantly improve the transfer efficiency of PCI buses with as few as one bridge since the master is not held in wait states while the bridge completes the transaction on the destination bus. Furthermore, Delayed Transactions allow more combinations of transactions crossing the bridge in opposite directions to run concurrently than would otherwise be possible thus avoiding potential starvation problems and improving efficiency.

Only non-posted transactions can be completed as Delayed Transactions by a bridge. These include I/O Read, I/O Write, Configuration Read, Configuration Write, Memory Read, Memory Read Line, and Memory Read Multiple. Memory Write and Memory Write and Invalidate transactions are postable and, therefore, cannot be completed as Delayed Transactions.

To complete a transaction using Delayed Transaction termination, a bridge must latch the following information:

- address
- command
- byte enables
- address and data parity
- **REQ64#** (if a 64-bit transfer)

For write transactions completed using Delayed Transaction termination, a bridge must also latch data from byte lanes for which the byte enable is asserted and may optionally latch data from byte lanes for which the byte enable is deasserted. **LOCK#** must also be latched for transactions flowing downstream. Refer to Section 5.4. for additional requirements when completing a Delayed Transaction as part of a locked operation.

After latching the required information, the bridge terminates the transaction on the originating bus with Retry. Once ordering requirements have been satisfied (see Section 5.5.), and the arbiter has granted the destination bus to the bridge, the bridge begins the process of executing the transaction on the destination bus. If the Delayed Request is a read, the bridge obtains the requested data and completion status. If the Delayed Request is a write, the bridge delivers the write data and obtains the completion status. Completing the Delayed Request on the destination bus produces a Delayed Completion which consists of the latched information of the Delayed Request and the completion status (and data if a read request). The bridge stores the Delayed Completion until the master repeats the initial request.

The bridge differentiates between transactions (by the same or different masters) by comparing the current transaction with information latched previously (for both Delayed Requests and Delayed Completions). When the Parity Error Response bit (bit 6 of the Command Register for the primary bus and bit 0 of the Bridge Control register for the secondary bus) is cleared, the

bridge ignores the address and data parity latched previously when doing the comparison. The byte enables may optionally be ignored in the comparison if the master is reading from a prefetchable location, even though the master is required to repeat the transaction with the same byte enables. If the compare matches a Delayed Request (already enqueued), but the bridge is not ready to complete the request, the bridge does not enqueue the request again but simply terminates the transaction with Retry. If the compare matches a Delayed Completion and the bridge is ready to complete the request, the bridge responds by signaling the status and provides the data if a read transaction.

The master must repeat the transaction exactly as the original request; otherwise, the bridge will assume it is a new transaction (since the agent cannot distinguish masters). Two masters could request the exact same transaction and the bridge cannot and need not distinguish between them and will simply complete the Delayed Transaction with the first master to repeat the transaction after the completion on the destination bus by the bridge.

A bridge is permitted to enqueue one or more Delayed Requests at a time. If a bridge enqueues multiple Delayed Requests, the order in which it attempts them on the destination bus is independent of the order in which they were originally attempted on the originating bus. Furthermore, the order in which the transactions ultimately complete on the originating bus is independent of the order in which they were attempted on either bus and the order in which they completed on the secondary bus. (Refer to Section 5.5., for restrictions with respect to posted memory writes.)

While completing a Delayed Request, the bridge may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a locked transaction. However, the bridge cannot require a Delayed Transaction to complete on the originating bus before accepting the memory write data from a master on that bus; otherwise, a deadlock may occur. Furthermore, the bridge cannot indefinitely terminate a memory write with Retry on one bus because it is waiting to run a previously enqueued Delayed Request on the other bus, or because it is waiting for a master to take a previously enqueued Delayed Completion on either bus. Refer to Section 5.5. and Section 5.6.3. for more details.

5.3.1. Discarding a Delayed Request

Since a Delayed Request is only a request and not really a transaction, the bridge is allowed to discard a Delayed Request from the time it is enqueued until it has been attempted on the destination bus. Once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus and cannot be discarded. The bridge is allowed to present other requests. But if it attempts more than one request, the bridge must continue to repeat all requests that have been attempted unconditionally until they complete. The repeating of the requests is not required to be equal, but is required to be fair.

The bridge is allowed to discard Delayed Completions in only two cases. The first case is if the Delayed Completion is a read of a prefetchable region (or the command was Memory Read Line or Memory Read Multiple). The second case is for all Delayed Completions (read or write, prefetchable or not) if the master has not repeated the request before the Discard Timer interval is exceeded. When this occurs, the device is required to discard the Delayed Completion; otherwise, a deadlock may occur.

5.3.2. Discarding a Delayed Completion

The Discard Timer for masters on the primary bus is selectable to expire either within 2^{15} clocks or 2^{10} clocks depending on the state of bit 8 in the Bridge Control register. The Discard Timer for masters on the secondary bus is similarly controlled by bit 9 in the Bridge Control register. The longer value is the default and should be adequate for most systems. However, some classes of devices designed before Delayed Transactions were introduced into the *PCI Local Bus Specification* may encounter situations in which a transaction terminated with Retry is not repeated. If this situation occurs frequently enough that the longer Discard Timer value causes a performance problem, then the shorter time can be selected. When the Discard Timer interval is exceeded on either the primary interface or the secondary interface, the bridge must set the Discard Timer Status bit (bit 10) in the Bridge Control register. In addition, the bridge must assert **SERR#** on the primary interface if enabled to do so by the Discard Timer SERR# Enable bit (bit 11) in the Bridge Control register and the SERR# Enable bit in the Command register.

While completing a Delayed Request, the bridge may, from time to time, terminate a memory write transaction with Retry while temporary internal conflicts are being resolved; for example, when all the memory-write data buffers are full or during a locked transaction. However, the bridge cannot require a Delayed Transaction to complete on the originating bus before accepting the memory write data from a master on that bus; otherwise, a deadlock may occur. Furthermore, the bridge cannot indefinitely terminate a memory write with Retry on one bus because it is waiting to run a previously enqueued Delayed Request on the other bus, or because it is waiting for a master to take a previously enqueued Delayed Completion on either bus. Refer to Section 5.5. and Section 5.6.3. for more details.

5.4. Exclusive Access Transactions

A PCI-to-PCI bridge is only allowed to propagate an exclusive access transaction from its primary to its secondary interface and never allowed to initiate an exclusive access of its own initiative. A PCI-to-PCI bridge is required to ignore **LOCK#** when acting as a target on its secondary interface. A PCI-to-PCI bridge adheres to the **LOCK#** usage requirements defined in the *PCI Local Bus Specification*. This section describes additional requirements for propagating an exclusive access across a PCI-to-PCI bridge.

The first transaction of a lock operation must be a read transaction and is completed by the bridge using Delayed Transaction termination. When a downstream read transaction with **LOCK#** is successfully queued as a Delayed Request (Delayed Lock-Request) on the primary interface, the bridge enters a *target-lock* state even though the master is terminated with Retry. While in the target-lock state, the bridge terminates with Retry all transactions on the primary interface.

When the downstream Delayed Lock-Request is queued on the primary interface, it is forwarded to the secondary interface exactly the same as an unlocked Delayed Request (i.e., all ordering rules are obeyed. See Section 5.5.). The bridge must follow the requirements for initiating an exclusive access before attempting the Delayed Lock-Request.

5.4.1. Delayed Lock-Request Error

If the Delayed Lock-Request completes on the secondary interface with Master-Abort or Target-Abort, the bridge has not successfully established exclusive access of the target device and does not establish control of the secondary interface **LOCK#** resource. In this case, the bridge is allowed to respond to subsequent upstream transactions normally (i.e., does not unconditionally terminate with Retry upstream posted writes or upstream Delayed Transactions). When the Delayed Lock-Request terminates with Master-Abort or Target-Abort on the secondary interface, the error information is retained in the Delayed Lock-Completion that is produced.

The Delayed Lock-Completion is returned to the primary interface exactly the same as an unlocked Delayed Completion (i.e., all ordering rules are obeyed; refer to Section 5.5.). When the Delayed Lock-Completion is returned to the originating master, the bridge is required to signal a Target-Abort to the originating master on the primary interface. When the bridge signals Target-Abort to the originating master, the primary interface of the bridge transitions from the target-lock state to an unlocked state (lock is not established). In this case, the master has not successfully established exclusive access of the target device, and it does not establish control of the primary interface **LOCK#** resource.

5.4.2. Normal Completion

If the Delayed Lock-Request completes without error on the secondary interface (normal completion or disconnect), then the bridge has successfully established exclusive access of the target device and ownership of the secondary interface **LOCK#** resource. The secondary interface of the bridge enters a locked state, and the bridge must terminate with Retry all subsequent upstream posted writes. The bridge may optionally terminate with Retry upstream Delayed Transactions while the secondary interface is in the locked state.

The Delayed Lock-Completion is returned to the primary interface exactly the same as an unlocked Delayed Completion (i.e., all ordering rules are obeyed). When the Delayed Lock-Completion is returned to the originating master, the bridge signals normal completion (including disconnect). At this point, the primary interface transitions from the target-lock state to a full-lock state (lock is established across the bridge). In this case, the master has successfully established exclusive access of the target device and ownership of the primary interface **LOCK#** resource.

Once **LOCK#** is established between the originating master and the target device, both the primary and secondary interfaces of the bridge remain in their locked states. Both interfaces remain in the locked state until the originating master releases **LOCK#** ownership on the primary interface. The originating master relinquishes **LOCK#** ownership when it completes the desired sequence of exclusive operations or the bridge signals a Target-Abort to the master. When lock ownership is relinquished by the originating master, the primary interface of the bridge changes from the locked state to the unlocked state, the bridge relinquishes **LOCK#** ownership on the secondary bus, and the secondary interface of the bridge changes from the locked state to the unlocked state.

5.5. Ordering Requirements

Appendix E of the *PCI Local Bus Specification* specifies the ordering requirements for PCI-PCI bridges. Sections of Appendix E are duplicated here for convenience. For a full discussion of PCI ordering requirements, refer to the *PCI Local Bus Specification*.

Summary of PCI Ordering Requirements

Following is a summary of the general PCI ordering requirements presented in the *PCI Local Bus Specification*.

General Requirements

1. The order of a transaction is determined when it completes. Transactions terminated with Retry are only requests and can be handled by the bridge in any order.
2. Memory writes can be posted in both directions in a bridge. I/O and Configuration writes are not posted. (I/O writes can be posted in the host bridge, but some restrictions apply.) Read transactions (Memory, I/O, or Configuration) are not posted.
3. Posted memory writes moving in the same direction through a bridge will complete on the destination bus in the same order they complete on the originating bus.
4. Write transactions crossing a bridge in opposite directions have no ordering relationship.
5. A read transaction must push ahead of it through the bridge any posted writes originating on the same side of the bridge and posted before the read. Before the read transaction can complete on its originating bus, it must pull out of the bridge any posted writes that originated on the opposite side and were posted before the read command completes on the read-destination bus.
6. A bridge can never make the acceptance (posting) of a memory write transaction as a target contingent on the prior completion of a non-locked transaction as a master on the same bus. Otherwise, a deadlock may occur. Bridges are allowed to refuse to accept a memory write for temporary conditions which are guaranteed to be resolved with time. A bridge can make the acceptance of a memory write transaction as a target contingent on the prior completion of locked transaction as a master only if the bridge has already established a locked operation with its intended target.

The following is a summary of the PCI ordering requirements specific to Delayed Transactions presented in the *PCI Local Bus Specification*.

Delayed Transaction Requirements

1. A target that uses Delayed Transactions may be designed to have any number of Delayed Transactions outstanding at one time.
2. Only non-posted transactions can be handled as Delayed Transactions.
3. A master must repeat any transaction terminated with Retry since the target may be using a Delayed Transaction.

4. Once a Delayed Request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, it is only a request and may be discarded at anytime.
5. A Delayed Completion can only be discarded when it is a read from a prefetchable region, or if the master has not repeated the transaction in 2^{15} or 2^{10} clocks.
6. A target must accept all memory writes addressed to it even while completing a request using Delayed Transaction termination.
7. Delayed Requests and Delayed Completions have no ordering requirements with respect to themselves.
8. Delayed Completions must be given an opportunity to pass Delayed Requests.
9. Only a Delayed Write Completion can pass a Posted Memory Write. A Posted Memory Write must be given an opportunity to pass everything except another Posted Memory Write.
10. A single master may have any number of outstanding requests terminated with Retry. However, if a master requires one transaction to be completed before another, it cannot attempt the second one on PCI until the first one has completed.

Ordering of Requests

A transaction is considered to be a *request* when it is presented on the bus. When the transaction is terminated with Retry, it is still considered a request. A transaction becomes *complete* or a *completion* when data actually transfers (or is terminated with Master-Abort or Target-Abort). The following discussion will refer to a transaction as being a request or completion depending on the success of the transaction.

A transaction that is terminated with Retry has no ordering relationship with any other access. Ordering of accesses is only determined when an access completes (transfers data). For example, four masters A, B, C, and D reside on the same bus segment and all desire to generate an access on the bus. For this example, each agent can only request a single transaction at a time and will not request another until the current access completes. The order in which transactions complete are based on the algorithm of the arbiter and the response of the target, not the order in which each agent's **REQ#** signal was asserted. Assuming that some requests are terminated with Retry, the order in which they complete is independent of the order they were first requested. By changing the arbiter's algorithm, the completion of the transactions can be any sequence (i.e., A, B, C, and then D or B, D, C, and then A, and so on). Because the arbiter can change the order in which transactions are requested on the bus, and, therefore, the completion of such transactions, the system is allowed to complete them in any order it desires. This means that a request from any agent has no relationship with a request from any other agent. The only exception to this rule is when **LOCK#** is used, which is described later.

Take the same four masters (A, B, C, and D) used in the previous paragraph and integrate them onto a single piece of silicon (a multi-function device). For a multi-function device, the four masters operate independent of each other, and each function only presents a single request on the bus for this discussion. The order their requests complete is the same as if they where separate agents and not a multi-function device, which is based on the arbitration algorithm. Therefore, multiple requests from a single agent may complete in any order, since they have no relationship to each other.

Another device, not a multi-function device, has multiple internal resources that can generate transactions on the bus. If these different sources have some ordering relationship, then the device must ensure that only a single request is presented on the bus at any one time. The agent must not attempt a subsequent transaction until the previous transaction completes. For example, a device has two transactions to complete on the bus, Transaction A and Transaction B and A must complete before B to preserve internal ordering requirements. In this case, the master cannot attempt B until A has completed.

The following example would produce inconsistent results if it were allowed to occur. Transaction A is to a flag that covers data, and Transaction B accesses the actual data covered by the flag. Transaction A is terminated with Retry, because the addressed target is currently busy or resides behind a bridge. Transaction B is to a target that is ready and will complete the request immediately. Consider what happens when these two transactions are allowed to complete in the wrong order. If the master allows Transaction B to be presented on the bus after Transaction A was terminated with Retry, Transaction B can complete before Transaction A. In this case, the data may be accessed before it is actually valid. The responsibility to prevent this from occurring rests with the master, which must block Transaction B from being attempted on the bus until Transaction A completes. A master presenting multiple transactions on the bus must ensure that subsequent requests (that have some relationship to a previous request) are not presented on the bus until the previous request has completed. The system is allowed to complete multiple requests from the same agent in any order. When a master allows multiple requests to be presented on the bus without completing, it must repeat each request independent of how any of the other requests complete.

Ordering of Delayed Transactions

A Delayed Transaction progresses to completion in three phases:

1. Request by the master
2. Completion of the request by the target
3. Completion of the transaction by the master

During the first phase, the master generates a transaction on the bus, the target decodes the access, latches the information required to complete the access, and terminates the request with Retry. The latched request information is referred to as a Delayed Request. During the second phase, the target independently completes the request on the destination bus using the latched information from the Delayed Request. The result of completing the Delayed Request on the destination bus produces a Delayed Completion which consists of the latched information of the Delayed Request and the completion status (and data if a read request). During the third phase, the master successfully re-arbitrates for the bus and reissues the original request. The target decodes the request and gives the master the completion status (and data if a read request). At this point, the Delayed Completion is retired and the transaction has completed.

The number of simultaneous Delayed Transactions a bridge is capable of handling is limited by the implementation and not by the architecture. Table 5-2 represents the ordering rules when a bridge in the system is capable of allowing multiple transactions to proceed in each direction at the same time. Each column of the table represents an access that was accepted by the bridge earlier, while each row represents a transaction just accepted. The contents of the box indicate what ordering relationship the second transaction must have to the first.

PMW - *Posted Memory Write* is a transaction that has completed on the originating bus before completing on the destination bus and can only occur for Memory Write and Memory Write and Invalidate commands.

DRR - *Delayed Read Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Read, Configuration Read, Memory Read, Memory Read Line, or Memory Read Multiple commands. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes on the destination bus. Before it is attempted on the destination bus, the DRR is only a request and may be discarded at any time to prevent deadlock or improve performance since the master must repeat the request later.

DWR - *Delayed Write Request* is a transaction that must complete on the destination bus before completing on the originating bus and can be an I/O Write or Configuration Write commands. Note: Memory Write and Memory Write and Invalidate transactions must be posted (PMW) and not be completed as DWR. As mentioned earlier, once a request has been attempted on the destination bus, it must continue to be repeated until it completes. Before it is attempted on the destination bus, the DWR is only a request and may be discarded at any time to prevent deadlock or improve performance since the master must repeat the request later.

DRC - *Delayed Read Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus to complete. The DRC contains the data requested by the master and the completion status (normal, Master-Abort, Target-Abort, parity error, etc.).

DWC - *Delayed Write Completion* is a transaction that has completed on the destination bus and is now moving toward the originating bus. The DWC does not contain the data of the access but only status of how it completed (Normal, Master-Abort, Target-Abort, parity error, etc.). The write data has been written to the specified target.

No - indicates that the subsequent transaction is not allowed to complete before the previous transaction to preserve ordering in the system. The four No boxes found in column 2 prevent PMW data from being passed by other accesses and thereby maintain a consistent view of data in the system.

Yes - indicates that the subsequent transaction must be allowed to complete before the previous one or a deadlock can occur.

When blocking occurs, the PMW is required to pass the DRC or the DWC. If the master continues attempting to complete Delayed Requests, it must be fair in attempting to complete the PMW. [DN]

Yes/No - indicates that the bridge designer may choose to allow the subsequent transaction to complete before the previous transaction or not. This is allowed since there are no ordering requirements to meet or deadlocks to avoid. How a bridge designer chooses to implement these boxes may have a cost impact on the bridge implementation or performance impact on the system.

Table 5-2: Ordering Rules for a Bridge

Row pass Col.?	PMW (Col 2)	DRR (Col 3)	DWR (Col 4)	DRC (Col 5)	DWC (Col 6)
PMW (Row 1)	No[1]	Yes[5]	Yes[5]	Yes[7]	Yes[7]
DRR (Row 2)	No[2]	Yes/No	Yes/No	Yes/No	Yes/No
DWR (Row 3)	No[3]	Yes/No	Yes/No	Yes/No	Yes/No
DRC (Row 4)	No[4]	Yes[6]	Yes[6]	Yes/No	Yes/No
DWC (Row 5)	Yes/No	Yes[6]	Yes[6]	Yes/No	Yes/No

Rule 1 - A subsequent PMW cannot pass a previously accepted PMW. (Col 2, Row 1)

Posted Memory write transactions must complete in the order they are received. If the subsequent write is to the flag that covers the data, the Consumer may use stale data if write transactions are allowed to pass each other.

Rule 2 - A read transaction must push posted write data to maintain ordering. (Col 2, Row 2)

For example, a memory write to a location and followed by an immediate memory read of the same location returns the new value (refer to the Special Considerations Section of the *PCI Local Bus Specification*, for possible exceptions). Therefore, a memory read cannot pass posted write data. An I/O read cannot pass a PMW, because the read may be ensuring the write data arrives at the final destination.

Rule 3 - A non-postable write transaction must push posted write data to maintain ordering. (Col 2, Row 2)

A Delayed Write Request may be the flag that covers the data previously written (PMW), and, therefore, the flag cannot pass the data that it potentially covers. (Col 2, Row 3)

Rule 4 - A read transaction must pull write data back to the originating bus of the read transaction. (Col 2, Row 4)

For example, the read of a status register of the device writing data to memory must not complete before the data is pulled back to the originating bus. Otherwise, stale data may be used.

Rule 5 - A Posted Memory Write must be allowed to pass a Delayed Request (read or write) to avoid deadlocks. (Col 3 and Col 4, Row 1)

Referring to Figure 5-1, bridge Y (using Delayed Transactions) is between bridges X and Z (designed to a previous version of this specification and not using Delayed Transactions). Consider the following sequence of events:

- Master 1 initiates a read to Target 1 that is forwarded through bridge X and is queued as a Delayed Request in bridge Y.
- Master 3 initiates a read to Target 3 that is forwarded through bridge Z and is queued as a Delayed Request in bridge Y.
- After Masters 1 and 3 are terminated with Retry, Masters 2 and 4 begin long memory write transactions addressing Targets 2 and 4 respectively, which are posted in the write buffers of bridges X and Z respectively.

When bridge Y attempts to complete the read in either direction, bridges X and Z must flush their posted write buffers before allowing the Read Request to pass through it. If the posted write buffers of bridges X and Z are larger than those of bridge Y, bridge Y's buffers will fill. Bridge Y cannot discard the read request since it has been attempted, and it cannot accept any more write data until the read in the opposite direction is completed. Since this condition exists in both directions, neither DRR can complete because the other is blocking the path. Therefore, the PMW data is required to pass the DRR when the DRR blocks forward progress of PMW data.

The same condition exists when a DWR sits at the head of both queues, since some old bridges also require the posting buffers to be flushed on a non-posted write cycle.

Rule 6 – Delayed Completion (read and write) must be allowed to pass Delayed Requests (read or write) to avoid deadlocks. (Cols 3 and 4, Rows 4 and 5)

Consider an application where the common PCI bus segment is on the secondary bus of bridge A and the primary bus for bridge B. If both bridges do not allow Delayed Completions to pass the Delayed Requests, neither can make progress.

For example, suppose bridge A's request to bridge B completes on bridge B's secondary bus, and bridge B's request completes on bridge A's primary bus. Bridge A's completion is now behind bridge B's request and bridge B's completion is behind bridge A's requests. Therefore, Delayed Completions must be allowed to pass Delayed Requests.

Rule 7 - A Posted Memory Write must be allowed to pass a Delayed Completion (read or write) to avoid deadlocks. (Col 5 and Col 6, Row 1)

Consider another transaction scenario similar to that for Rule 5 (again refer to Figure 5-1). In this case, however, a DRC sits at the head of the queues in both directions of bridge Y at the same time. Again the old bridges (X and Z) contain posted write data from another master. The problem in this case, however, is that the read transaction cannot be *repeated* until all the posted write data is flushed out of the old bridge and the master is allowed to repeat its original request. Eventually, the new bridge cannot accept any more posted data because its internal buffers are full, and it cannot drain them until the DRC at the other end completes. When this condition exists in both directions, neither DRC can complete, because the other is blocking the path. Therefore, the PMW data is required to pass the DRC when the DRC blocks forward progress of PMW data.

The same condition exists when a DWC sits at the head of both queues.

Transactions that have no ordering constraints

Some Delayed Transactions (enqueued as Delayed Requests or Delayed Completions) have no ordering relationship with any other Delayed Requests or Delayed Completions. For example, the designer can (for performance or cost reasons) allow or disallow Delayed Requests to pass other Delayed Requests and Delayed Completions that were previously enqueued.

Delayed Requests can pass other Delayed Requests (Cols 3 and 4, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with other Delayed Requests, these four boxes are don't cares.

Delayed Requests can pass Delayed Completion (Col 5 and 6, Rows 2 and 3).

Since Delayed Requests have no ordering relationship with Delayed Completions, these four boxes are don't cares.

Delayed Completions can pass other Delayed Completion (Col 5 and 6, Rows 4 and 5).

Since Delayed Completions have no ordering relationship with other Delayed Completions, these four boxes are don't cares. Delayed Write Completions can pass posted memory writes or be blocked by them (Col 2, Row 5)
If the DWC is allowed to pass a PMW or if it remains in the same order, there is no deadlock or data inconsistencies in either case. The DWC data and the PMW data are moving in opposite directions, initiated by masters residing on different buses accessing targets on different buses.

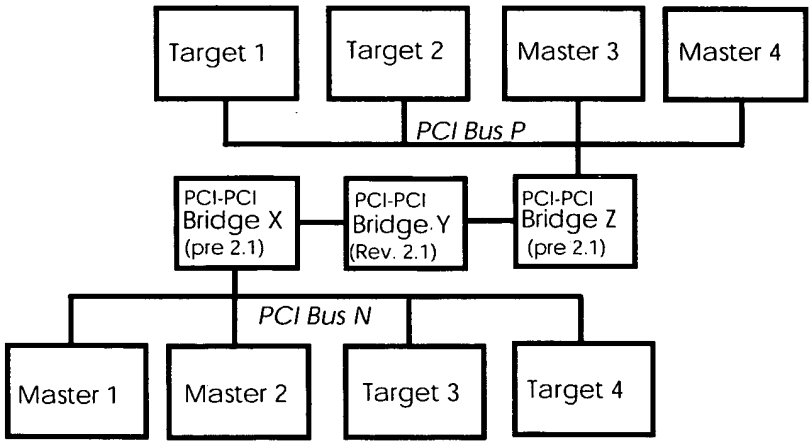

Figure 5-1: Example System with PCI-to-PCI Bridges

Delayed Transactions and LOCK#

The bridge is required to support **LOCK#** when a transaction is initiated on its primary bus (and is using the lock protocol), but is not required to support **LOCK#** on transactions that are initiated on its secondary bus. If a locked transaction is initiated on the primary bus and the bridge is the target, the bridge must adhere to the lock semantics defined by this specification. The bridge is required to complete (push) all PMWs (accepted from the primary bus) onto the secondary bus before attempting the lock on the secondary bus. The bridge may discard any requests enqueued (but not yet attempted on the secondary bus), allow the locked transaction to pass the enqueued requests, or simply complete all enqueued transactions before attempting the locked transaction on the secondary interface. Once a locked transaction has been enqueued by the bridge, the bridge cannot accept any other transaction from the primary interface until the lock has completed except for a continuation of the lock itself by the lock master. Until the lock is established on the secondary interface, the bridge is allowed to continue enqueuing transactions from the secondary interface, but not the primary interface. Once lock has been established on the secondary interface, the bridge cannot accept any posted write data moving toward the primary interface until **LOCK#** has been released (**FRAME#** and **LOCK#** deasserted on the same rising clock edge). (In the simplest implementation, the bridge does not accept any other transactions in either direction once lock is established on the secondary bus except for locked transactions from the lock master.) The bridge must complete previously enqueued PMW, DRC, and DWC transactions moving toward the primary bus before allowing the locked access to complete on the originating bus.

Error Conditions

A bridge is free to discard data or status of a transaction that was completed using Delayed Transaction termination when the master has not repeated the request within 2^{10} PCI clocks (about 30 µs at 33 MHz). However, it is recommended that the bridge not discard the transaction until 2^{15} PCI clocks (about 983 µs at 33 MHz) after it acquired the data or status. The shorter number is useful in system where a master designed to a previous version of this specification frequently fails to repeat a transaction exactly as first requested. In this case, the bridge may be programmed to discard the abandoned Delayed Completion early and allow other transactions to proceed. Normally, however, the bridge would wait the longer time in case the repeat of the transaction is being delayed by another bridge or bridges designed to a previous version of this specification that did not support Delayed Transactions.

When this timer (referred to as the Discard Timer) expires, the device is required to discard the data; otherwise, a deadlock may occur.

> Note: When the transaction is discarded, data may be destroyed. This occurs when the discarded Delayed Completion is a read to a non-prefetchable region.

When the Discard Timer expires, the device may choose to report or ignore the error. When the data is prefetchable, it is recommended that the device ignore the error since system integrity is not affected. However, when the data is not prefetchable, it is recommended that the device report the error to its device driver since system integrity is affected. A bridge may assert **SERR#** since it does not have a device driver.

Illustrations of the Use of the Ordering Rules

To illustrate the use of the ordering rules, consider the following examples. Each example shows a sequence of transactions on each bus, with time advances from left to right. Generally no attempt is made to align the time scale of the two buses:

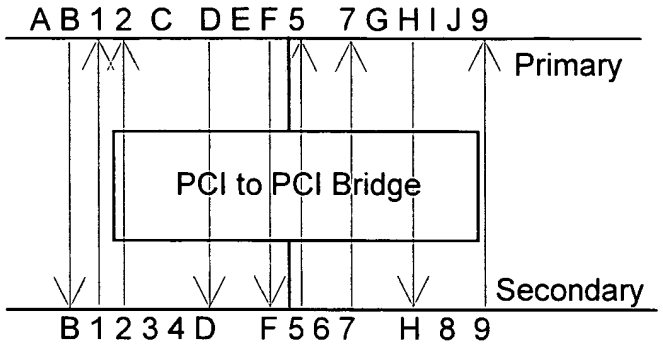

Figure 5-2: Transaction Ordering Example 1

In Figure 5-2, there are two streams of transactions, one on the primary bus and one on the secondary bus. Transactions that originate on the primary bus are designated by letters and complete in the sequence [A B C D E F G H I J]. Transactions that originate on the secondary bus are designated by numbers, and complete in the sequence [1 2 3 4 5 6 7 8 9]. Arrows are used to indicate that a transaction is forwarded from one bus to the other. The bridge forwards transaction B, D, F, and H from the primary bus to the secondary bus and transactions 1, 2, 5, 7, and 9 from the secondary bus to the primary bus. The resulting sequence on the primary bus is [A B 1 2 C D E F 5 7 G H I J 9] while the sequence on the secondary bus is [B 1 2 3 4 D F 5 6 7 H 8 9]. Notice that in this example, the arrows never cross, which indicates that in this example, the order of the transactions does not change independent of which bus the transaction initiates on or targets.

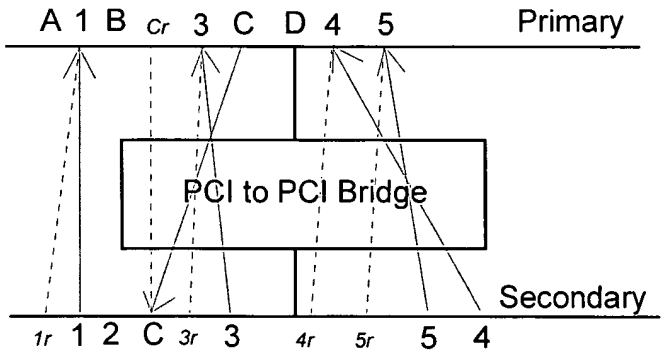

Figure 5-3: Transaction Ordering Example 2

To illustrate how the order of certain transactions can change when crossing a bridge, Figure 5-3 shows several Delayed Transactions. Initial enqueuing of the Delayed Request is designated in

italics with a small "*r*" and connected to the completed transaction on the destination bus with a dotted line. This illustrates when the master first requested the transactions but was terminated with Retry when the bridge enqueued the Delayed Request. As before, arrows indicate transactions that cross the bridge and point toward the destination bus. The head of the arrows shows when the transaction completed on the destination bus, and the tail of the arrow shows when the transaction completed on the originating bus. Delayed Transactions 3 and C are crossing the bridge in opposite directions. Although transactions C is terminated with Retry on the primary bus and completes on the secondary bus before transaction 3, it does not complete on the primary bus until after transaction 3. Similarly, Delayed Transactions crossing in the same direction can be reordered. Transactions 4 and 5 complete on the primary bus in the same order as they were terminated with Retry on the secondary bus, but they complete on the secondary bus in the opposite order. This reordering can occur because it depends upon such things as PCI bus arbitration sequence and timing of the master's request to repeat the transaction on the secondary bus relative to when it completed on the primary bus. In summary, the order in which Delayed Transactions start and complete with respect to each other on either bus is irrelevant.

The relevant transaction ordering for a bridge involves posted memory write transactions. As indicated in Table 5-2, the order relative to posted memory writes of most transactions moving in either direction must be maintained by the bridge.

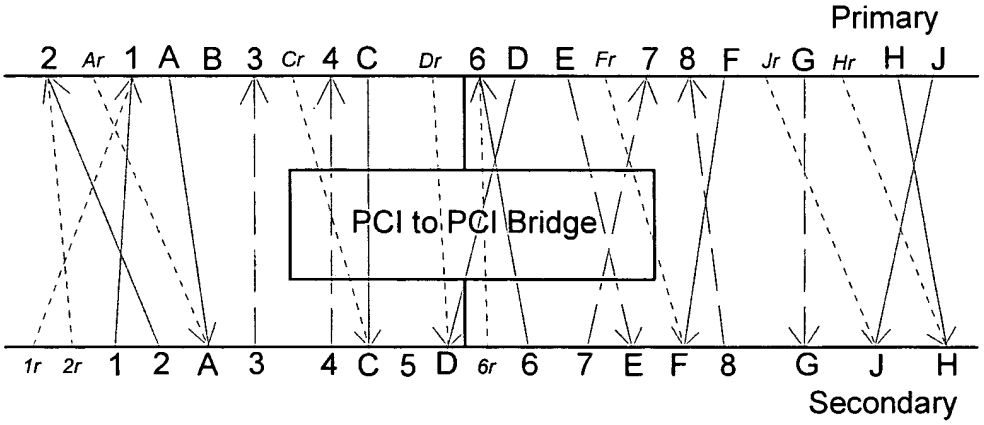

Figure 5-4: Transaction Ordering Example 3

Figure 5-4 illustrates some of the ordering cases involving posted memory writes. Transactions 3, 4, 7, 8, E, and G are posted memory writes, and are shown with long-dashed lines. The remaining transactions are all Delayed Transactions. Notice that posted memory write transactions crossing the bridge in the same directions complete on both busses in the same order, in this example [3 4 7 8] and [E G]. However, the order of a posted memory write with respect to another posted memory write crossing the bridge in the opposite direction is not important ([3 4 E 7 8 G] on primary side and [3 4 7 E 8 G] on secondary side).

No Delayed Request is allowed to pass a posted memory write going in the same direction, but posted memory writes are allowed to pass Delayed Requests. Delayed Request *Fr* is enqueued after posted memory write E and, therefore, Delayed Transaction F cannot complete on the secondary bus until after posted memory write E. However, posted memory write G can complete on the secondary bus before Delayed Transaction J, even though G completed on the primary bus after Delayed Request *Jr* was enqueued.

Figure 5-4 also illustrates that no Delayed Completion is allowed to pass a posted memory write going in the same direction, but posted memory writes are allowed to pass Delayed Completions. Since Delayed Transaction C completed after posted memory write 4 on the secondary bus, the Delayed Transaction cannot complete before the write on the primary bus. But even though posted memory write 8 completed on the secondary bus after Delayed Transaction F, the write can complete before the Delayed Transaction on the primary bus.

Besides maintaining a consistent view of write data, the ordering rules also avoid deadlocks. Unless a locked sequence is in progress, in general, the bridge must accept a posted memory write transaction addressing a target across the bridge, regardless of what other transactions preceded it on either side. The only exception is for temporary conditions like emptying the buffer of previous posted memory writes. The bridge may not continually terminate a memory write transaction with Retry while waiting for a non-locked transaction to complete in either direction.

5.6. Special Design Considerations

5.6.1. Read Starvation

Bridges designed to an earlier version of this specification do not implement Delayed Transactions, and typically do not meet the latency requirements of the *PCI Local Bus Specification* and can in normal operation starve masters on one side of the bus from fair access to the other side.

Consider, for example, the case where a single master on the secondary bus is executing a long string of write transactions addressing main memory, and during this time the CPU attempts to read from a target on the secondary bus. Since there are no other masters on the secondary bus, the writing master will quickly acquire the bus and post the first write transaction in the bridge. While the bridge is waiting to acquire the primary bus, the CPU is granted the bus and attempts to read from the secondary target. The ordering rules require the bridge to empty its posting buffer before allowing a read to complete, so it must Retry the CPU read. If the bridge does not implement Delayed Transactions, this CPU read is not enqueued and has made no forward progress through the bridge. When the bridge acquires the bus, it will execute its write and empty its posting buffer.

Since there are no other masters on the secondary bus in this example, it is quite possible for the same writing master to reacquire the secondary bus and refill the bridge's posting buffer before the CPU has a chance to repeat its read transaction. If this happens, then the sequence will repeat, starving the CPU until the master eventually finishes all of its write operations. Note that in some applications, such as a live video frame grabber, the write stream never stops.

This condition is avoided if the bridge executes the CPU read as a Delayed Transaction. In this case, when the CPU read is terminated with Retry, the bridge would enqueue a Delayed Read Request. If the secondary arbiter treats this request fairly with respect to secondary master requests for the bus, the CPU read would execute to completion on the secondary bus in between master writes to the bridge, even though the CPU had been terminated with Retry on the primary bus. The read data is then inserted between data of the write stream. The read data then reaches the primary bus where it is held until the CPU repeats the request. To complete the read

transaction the CPU need only reacquire the primary bus and repeat the transaction some time after the read data reaches the primary bus queue. Write data enqueued after the read data is accepted is allowed to pass the read completion if blocking occurs to avoid a deadlock.

5.6.2. Stale Data

It is the responsibility of the bridge to guarantee that any data provided to a master be current as of the time the master first attempted the read transaction. In general, this means that the bridge must discard the balance of any data prefetched on behalf of a master, but not taken when the master completed the transaction.

Implementation Note: Stale-Data Problems Caused by Not Discarding Prefetch Data

Suppose a CPU has two buffers in adjacent main memory locations. The CPU prepares a message for a bus master in the first buffer and then signals the bus master to pick up the message. When the bus master reads its message, a bridge between the bus master and main memory prefetches subsequent addresses including the second buffer location.

Sometime later, the CPU prepares a second message using the second buffer in main memory and signals the bus master to come and get it. If the intervening bridge has not flushed the balance of the previous prefetch, then when the master attempts to read the second buffer, the bridge may deliver stale data.

Similarly, if a device were to poll a location behind a bridge and the bridge did not flush the buffer after each read by the device, the device would never observe a new value for the polled location

The Special Design Considerations section of the *PCI Local Bus Specification*, describes another situation in which a master might see stale data. If two masters are polling the same location using the same address, command, and byte enables, and one of the masters also writes to the location, the next read by the writing master may read the value before the write rather than after it. This same problem can occur if the two masters are not sharing the same location, but use the same address, command, and byte enables, because one of the masters starts reading at a smaller address than the one it actually wants.

Since the bridge, in general, has no knowledge of which master actually is making the request, the bridge has no alternative but to supply the read completion data to the first master to repeat the identical read transaction. Although it is difficult to envision a real application that would behave this way, if one exists, then it is that application designer's responsibility to avoid the problem by doing a dummy read of the location or device after writing to it.

5.6.3. Deadlocks

The PCI ordering rules permit memory write transactions to be posted anywhere in the system, and require that from time to time those posted writes must be flushed before other transactions are allowed to complete so that all masters in the system will have a consistent view of data. As a result, deadlocks can occur in numerous cases if targets do not follow the ordering rules for accepting posted memory writes. In almost all cases, it is required that a target (including a

bridge) accept a posted memory write addressed to it regardless of what other non-locked transactions may have preceded it. The only exception is for conditions that are guaranteed to be resolved over time; for example, while all buffers are filled with previous posted memory write transactions.

For example, suppose there are two bridges connected hierarchically, with Master A and Target 1 at the top, and Master B and Target 2 at the bottom as shown in Figure 5-5. Further suppose that Master A executes a memory write to Target 2, which is posted in the upper bridge, and Master B executes a read from Target 1, which crosses the lower bridge and appears at the secondary interface of the upper bridge. To satisfy the ordering rules listed above, bridge X must first empty the posted memory write addressed to Target 2 before it can complete the read from Target 1 on PCI Bus P. Bridge Z is required to accept this posted memory write even if the read transaction has already been attempted and terminated with Retry on bridge Z's upper bus (the PCI bus connecting bridges X and Z). This is true regardless of whether the read is executed as a Delayed Transaction or not. If the lower bridge were to require its read to complete before accepting the posted memory write, the system would deadlock. The *PCI Local Bus Specification*, Appendix E includes other examples of deadlocks which can occur if targets (including bridges) don't accept posted memory writes.

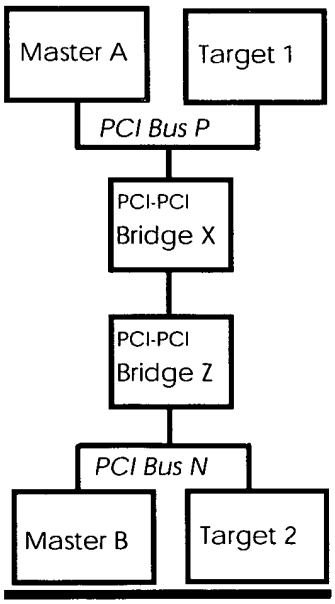

Figure 5-5: Deadlock Example

5.7. Combining Separate Writes Into a Single Burst Transaction

When a bridge forwards memory writes in either the memory mapped I/O or prefetchable memory address ranges, it is allowed to combine separate but sequential[5] memory writes into a single burst transfer (using linear increment addressing) provided the implied ordering is not changed. For example, separate writes to Dword 1, 2, and 4 can be combined and forwarded as a single burst (where the byte enables for Dword 3 are deasserted). However, separate writes to Dword 4, 3, and 1 cannot be combined into a burst but must be forwarded as three separate transactions in the same order as they were received.

Combining of I/O writes or configuration writes by a bridge is not allowed. Combining of memory writes by a bridge is optional. See the *PCI Local Bus Specification* for additional information on write combining.

5.8. Merging Separate Writes Into a Single Transaction

When a bridge forwards memory writes in a prefetchable memory address range, it is allowed to merge separate but sequential masked writes to one Dword address into a single data phase transfer, provided any byte location is written only once. For example, consider a sequence of separate byte writes to bytes 3, 1, 0, and 2 of a Dword. A bridge is allowed to merge these writes and forward them as a single write transaction. However, if the write sequence is byte 1, 1, 2, and 3, then the first write to byte 1 must be forwarded as a separate write transaction. The remaining byte writes could be forwarded as a single write with byte enables 1, 2, and 3 asserted and byte enable 0 deasserted.

Merging of I/O writes, configuration writes, or memory writes in the memory mapped I/O address range by a bridge is not allowed. Merging of memory writes in the prefetchable memory range by a bridge is optional. See the *PCI Local Bus Specification* for additional information on write merging.

5.9. Collapsing of Writes

When a bridge forwards write transactions, it cannot collapse sequential writes to the same address into a single transfer. Two sequential write transactions to the same address in which at least one byte enable has been asserted in both transactions must be forwarded as separate write transactions by a bridge.

Collapsing of writes of any type by a bridge is not allowed. See the *PCI Local Bus Specification* for additional information on write collapsing.

---

[5]The term sequential is used to indicate that the events (PCI transactions) occur in the order indicated without any other intervening transactions.